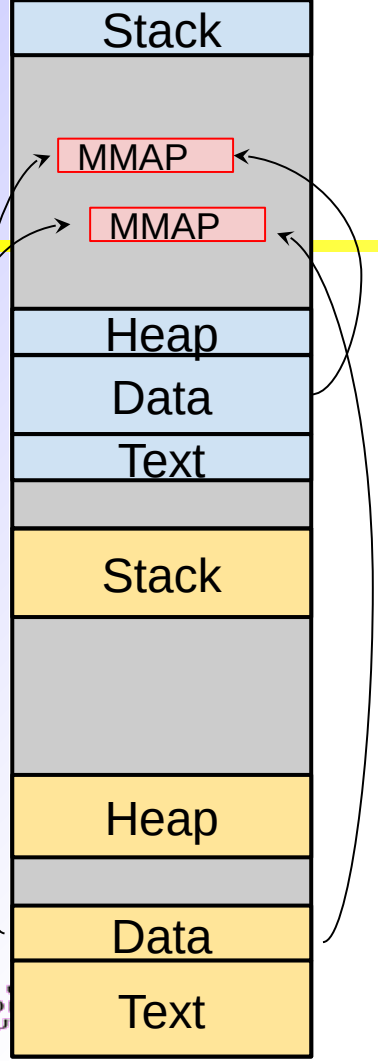
Sincronização

Problematização

Memória compartilhada

- Para 2 Processos
  - Através de mmap (ou através das chamadas shmem e shmap -- deprecated)
- Para 2 Threads
  - Por natureza

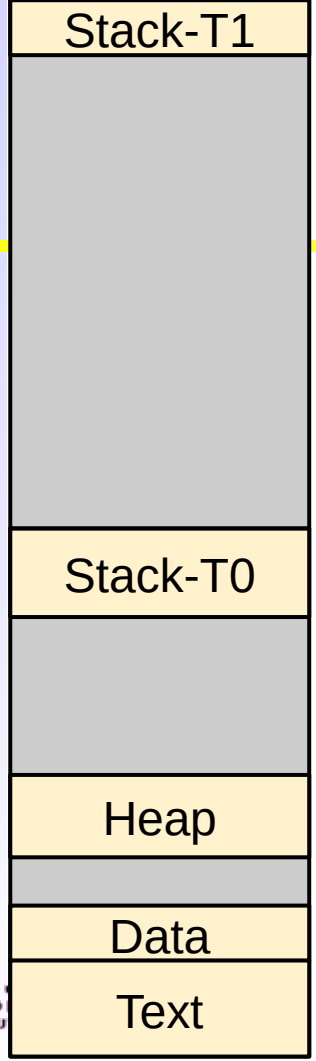


- Permite compartilhar regiões de memória entre 2 processos
  - MMaps do pai são herdados pelo filho no fork
- Vamos ver mais sobre mmap em outras aulas
- exemplo0.c
  exemplo0sem.c

Lembrando de PThreads

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
  - Cria nova Thread
  - Inicia a execução da Thread
  - Ponteiro para a Thread
  - Atríbutos
  - Ponteiro para função
  - Argumentos da Função

- int pthread_join(pthread_t thread, void **retval);
  - Espera a thread finalizar
  - Copia o valor de retorno para retval



- Já temos dados e o heap compartilhado
- Cada thread tem apenas um stack próprio
  - E valor dos registradores

Exemplo de condição de corrida

Código para incremento

`count++` em assembler:

```
a) MOV R1, $counter
b) INC R1
c) MOV $counter, R1
```

`count--` em assembler:

```
x) MOV R2, $counter
y) DEC R2
z) MOV $counter, R2
```

- Cada instrução é independente
- Interrupções podem ocorrer entre quaisquer duas instruções
  - Logo, trocas de contexto também podem ocorrer
- As sequências [a,b,c] e [x,y,z] podem ocorrer intercaladas

Falta de sincronização

```
a) MOV R1, $counter        R1 = 5
b) INC R1                  R1 = 6
x) MOV R2, $counter        R2 = 5
y) DEC R2                  R2 = 4
c) MOV $counter, R1        counter = R1 = 6
z) MOV $counter, R2        counter = R2 = 4
```

Troca de contexto

Troca de contexto

Condições de corrida

Dados e estruturas são acessados de forma concorrente

Resultado depende da ordem de execução dos processos

- Resultado é *indeterminado*

Necessidade de sincronização

Problema da seção crítica

Contexto

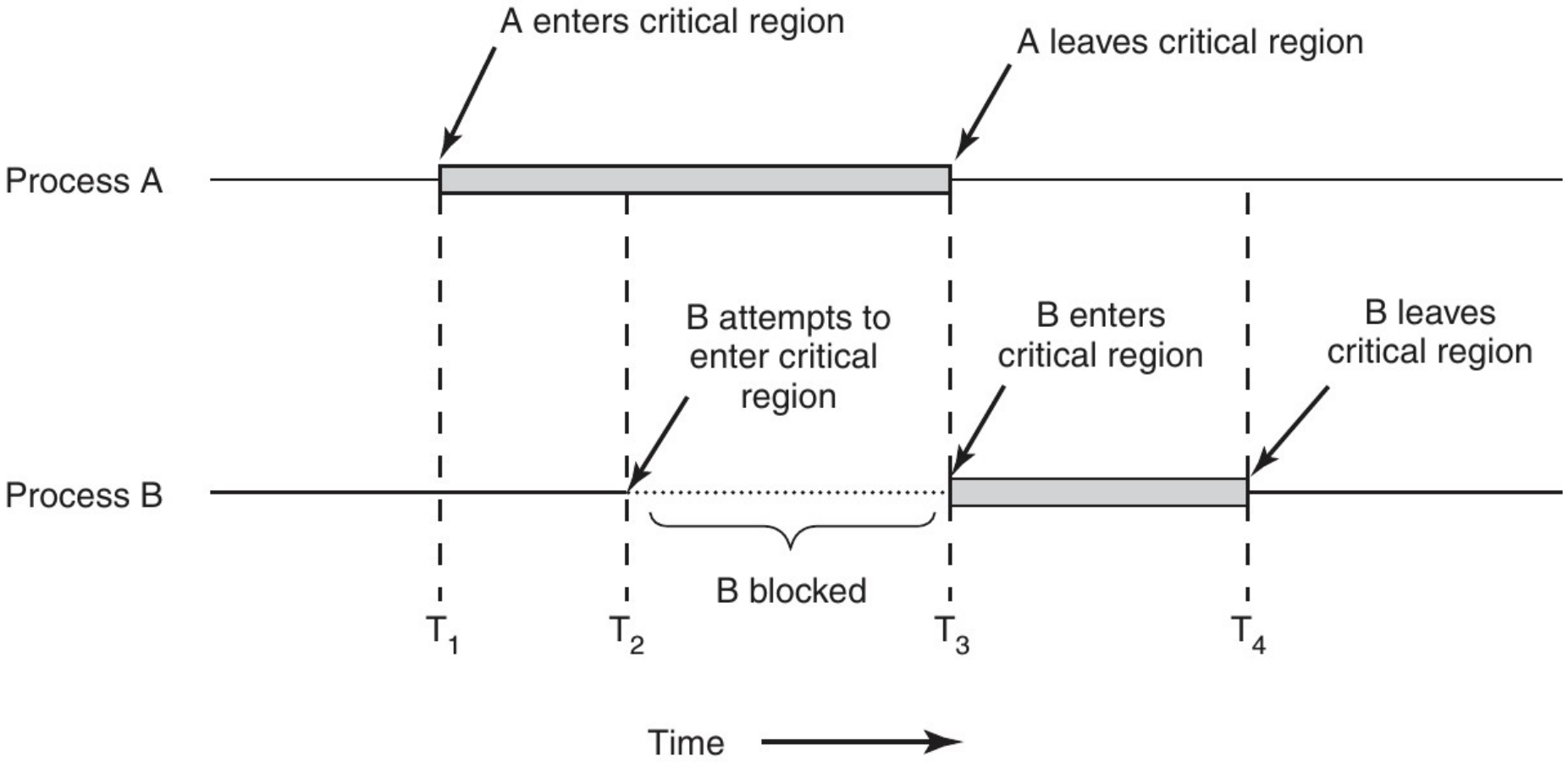
- Vários processos utilizam uma estrutura de dados compartilhada
- Cada processo tem um segmento de código onde a estrutura é acessada
- Processos executam a uma velocidade não nula (fazem progresso)
- O escalonamento e velocidade de execução são indeterminados

Garantir que apenas um processo acessa a estrutura por vez

Problema da seção crítica

Requisitos da solução:

- Exclusão mútua
  - Apenas um processo na seção crítica por vez
- Progresso garantido
  - Se nenhum processo está na seção crítica, qualquer processo que tente fazê-lo não pode ser detido indefinidamente
  - [Outra Forma de Pensar] Nenhum processo fora de sua região crítica pode bloquear outros
- Espera limitada
  - Se um processo deseja entrar na seção crítica, existe um limite no número de outros processos que entram antes dele


A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
T1
T2
T3
T4
Time

Controle de acesso à seção crítica

Considere dois processos, Pi e Pj

Processos podem compartilhar variáveis
para conseguir o controle de acesso

```
do {
    enter section
    // critical section
    leave section
    // remainder section
} while (1);
```

Solução Bazooka

- Desabilitar interrupções
- Parar o SO todo menos o processo que vai utilizar região crítica

Solução Bazooka

- Desabilitar interrupções
- Parar o SO todo menos o processo que vai utilizar região crítica
- Meio extrema, mas ok, funciona.
  - Pelo menos para a exclusão mútua
- Perdemos tudo que aprendemos de escalonamento
  - Além de outros problemas como processos fazendo tarefas de SO
- Vamos pensar em algo melhor

Tentativa 1

```
int turn; // variável de controle, compartilhada, inicializada para
i ou j

do {
    // id da thread na variável i
    while(turn != i);
    // critical section.
    turn = j;
    // remainder section
} while (1);
```

Solução de Peterson

```
bool flag[2]; // variável de controle, compartilhada
int turn;
do {
    flag[i] = TRUE;
    turn = j;
    while(flag[ j ] && turn == j);
    // critical section
    flag[i] = FALSE;
    // remainder section
} while (TRUE);
```

Solução de Peterson

- Funciona em sistemas de um processador apenas
- Hardwares modernos com vários cores não garantem sequência de operações
- Caches criam a oportunidade de discórdia
  - Cada CPU observa um valor diferente
- Instruções atômicas de hardware para resolver isto
  - e.g., TSL (Test-and-Set)
- Mais complicada com n processos

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
boolean test_and_set(boolean *target) {
   boolean old = *target;
   *target = 1;
   return old;
}
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
int lock = 0;
do {
    while(test_and_set(&lock));
    // critical section
    lock = 0;
    // remainder section
} while (1);
```

```
boolean test_and_set(boolean *target) {
   boolean old = *target;
   *target = 1;
   return old;
}
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
void swap(boolean *a, boolean *b) {
   boolean tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
int lock = 0; // compartilhada
do {
    boolean key = true;
    while(key) swap(&lock, &key);
    // critical section
    lock = 0;
    // remainder section
} while (1);
```

```
void swap(boolean *a, boolean *b) {
   boolean tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Esses algoritmos não satisfazem a esperada “limitada.”

Exclusão mútua justa - cada processo passa a vez para o próximo na fila

```
int lock = 0; int n; //n tem o número de processos
do {
     waiting[i] = 1;
     while(waiting[i] && test_and_set(&lock));
     // critical section
     waiting[i] = 0;
     j = (i + 1) % n;
     while((j != i) && !waiting[j])
          j = (j + 1) % n;
     if (j == i) { lock = 0; }
     else { waiting[j] = 0; }
     // remainder section
 } while (1);
```

UFMG - ICEx
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Exclusão mútua justa - cada processo passa a vez para o próximo na fila

```
int lock = 0; int n; //n tem o número de processos
do {
     waiting[i] = 1;
     while(waiting[i] && test_and_set(&lock));
     // critical section
     waiting[i] = 0;
     j = (i + 1) % n;
     while((j != i) && !waiting[j])
          j = (j + 1) % n;
     if (j == i) { lock = 0; }
     else { waiting[j] = 0; }
     // remainder section
 } while (1);
```

```
// Caso alguém, j, waiting → libere.
// Se a Pi que estava na RC voltar
// por cima, o lock=1 garante que Pi
// não entre na SC antes de Pj. Lock só
// fica igual a 0 quando ninguém espera
```

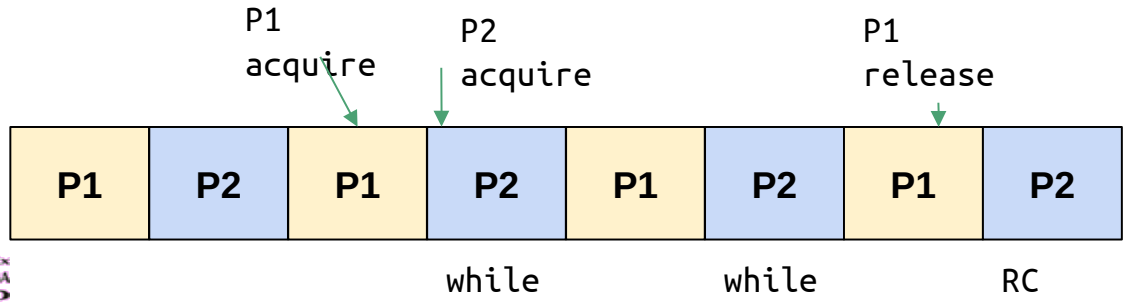
Primitivas

[Eficiência] Busy wait; espera ocupada; spin locks

- Vamos quebrar a solução de Peterson (2 threads/processos) ou a Bounded-Wait Mutex (n threads/processos) em 2 chamadas

```
acquire() { . . . } // antes da RC
release() { . . . } // depois da RC
```

Assumindo round-robin

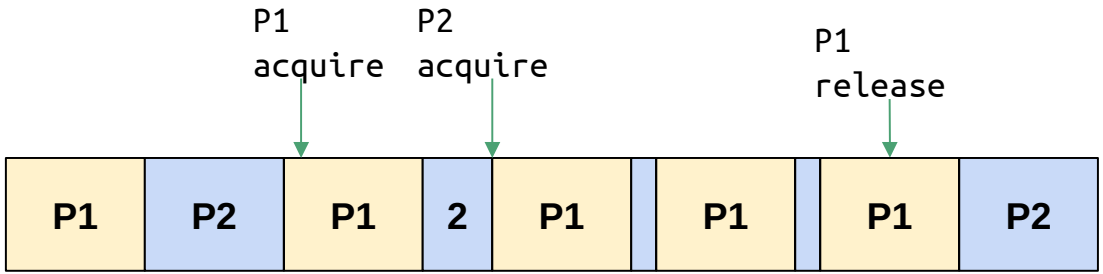


Melhorando o busy wait com yield

```
int queue[2] = {0, 0}; // variável de controle,
compartilhada
int turn = i;
do {
    queue[i] = 1;
    turn = j;
    while(queue[j] && turn == j) yield();
    // critical section
    queue[i] = 0;
    // remainder section
} while (1);
```

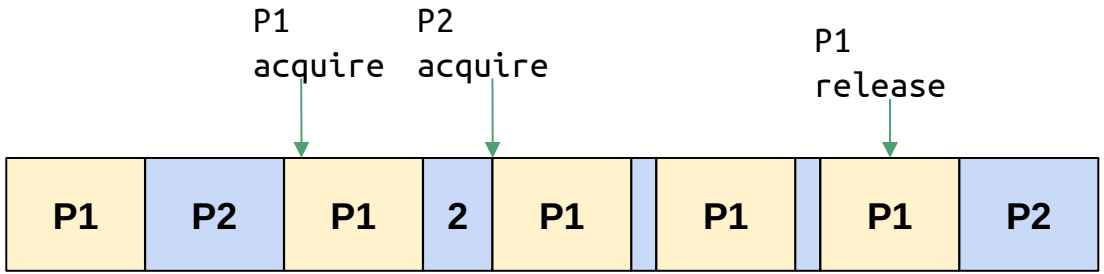
yield

- Libera a CPU



yield

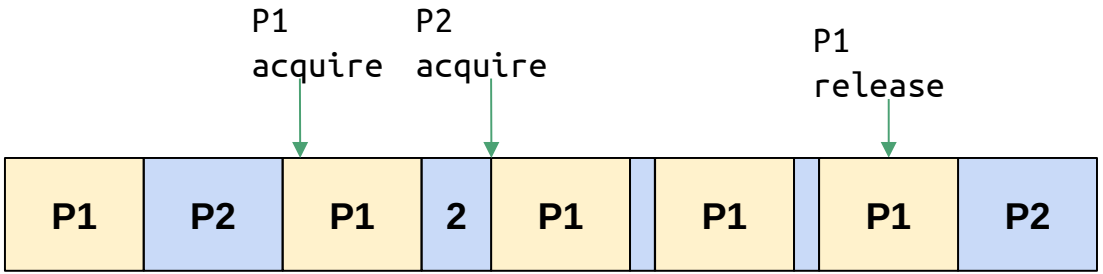
- Libera a CPU



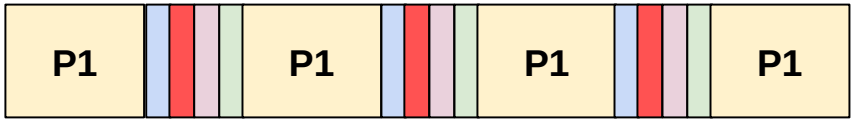
- Melhora só que com muitos processos...

yield

- Libera a CPU



- Melhora só que com muitos processos...



UFMG - ICEx
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Problema de prioridades

- Ainda fazendo uso das primitivas de aquire e release.
  Qual o problema do código abaixo?

```
acquire() { . . . } // antes da RC
release() { . . . } // depois da RC
```

- Vamos supor que um escalonador sempre escalone P1 antes de P2
  - P1 tem maior prioridade que sempre é fixa

```
P1: read_disk() // P1 vai para waiting
P2: acquire()   // P2 entra na RC
P1: acquire()   // P1 vai para waiting
```

yield

- Ajuda no uso de CPU
  - Menos desperdício
- Resolve a inversão de prioridades?

yield

- Ajuda no uso de CPU
  - Menos desperdício
- Resolve o problema de prioridades?
  - Não!

Removendo o busy wait

- SO pode ajudar
- Primitivas de sleep/wakeup
  - sleep → coloca o processo em waiting
  - wakeup → coloca o processo em ready

Semáforos

Primitivas de alto nível oferecidas pelo sistema operacional ou linguagem

Conceitualmente, semáforo é uma variável inteira acessível por duas operações atômicas

Funciona com qualquer número de processos. Conceito com busy wait abaixo

```
wait(int *s) {
   while(*s <= 0);
   (*s)--;
}
// acquire, down, lock
```

```
signal(int *s) {
   (*s)++;
}

// release, up, unlock
```

Intuição de semáforos

```
typedef struct { . . . } process_t;
typedef struct {
    int value;
    process_t *list;
} sem_t;

void wait(process_t *me, sem_t *s) {
   s->value--;
   if(s->value < 0) {
      add(me, s->list);
      sleep(me); // vai para waiting
   }
}
```

```
void signal(sem *s) {
   s->value++;
   if(s->value <= 0) {
      p = remove(s->list);
      wakeup(p);
   }
}
```

O semáforo em si tem condições de corrida

- Value e a lista de processos
- Como garantir que os mesmos são implementados corretamente?

```
typedef struct {
    int value;
    process_t *list;
} sem_t;
```

O semáforo em si tem condições de corrida

- Value e a lista de processos
- Como garantir que os mesmos são implementados corretamente?

```
typedef struct {
    int value;
    process_t *list;
} sem_t;
```

- Pequenos trechos de código em que podemos:
  - Implementar uma espera ocupada

    ou

  - Desabilitar as interrupções

Semáforo binário com sleep

```
// structs e funções auxiliares
typedef struct { . . . } process_t;
typedef struct {
  int flag;
  int guard;
  process_t *list;
} bin_sem_t;

void bin_init(bin_sem_t *m) {
  m->flag = 0;
  m->guard = 0;
  list_init(m->list);
}
void list_add(process_t *list) {
  // . . . Lista fifo insere
  // sem interrupção
}

void list_remove(process_t *list) {
  // . . . Lista fifo remove
  // sem interrupção
}
```

```
// código de wait/signal
void bin_wait(bin_sem_t* m) {
  while (test_and_set(&m->guard));
  if (m->flag == 0) {
    m->flag = 1;
    m->guard = 0; // libera loop
  } else {
    process_t *me = getself();
    list_add(me, s→list);
    m->guard = 0; // libera loop
    sleep(me);    // waiting
  }
}
void bin_signal(bin_sem_t* m) {
  while (test_and_set(&m->guard));
  if (empty(m->list))
    m->flag = 0; // ninguém esperando
  else {
    process_t *p = list_remove(s->list);
    wakeup(p);
  }
  m->guard = 0;  // libera loop
}
```

UFMG - ICEx
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Estendendo o semáforo

- Construir outras primitivas com base neste
  - Mutexes
  - Barreiras
  - Thread pools
  - Monitores
  - Canais
  - ...
- Como implementar o semáforo de tamanho n com base no binário?

Semáforo de Tamanho N

```
// o semáforo faz uso de 2
// semáforos binários.
// 1 para cuidar da RC do value
// outro para realmente esperar
typedef struct {
    int value;
    // cuida do value
    binsem_t *mutex = 1;
    // usado para esperar
    binsem_t *wait = 1;
} sem_t;

void sem_init(sem_t *s,
              int value) {
  bin_init(s->mutex);
  bin_init(s->wait);
  s->value = value;
}
```

```
void wait(sem_t *s) {
   bin_wait(s->mutex);
   s->value--;
   if(s->value < 0) {
      bin_signal(s->mutex);
      bin_wait(s->wait);
   } else {
      bin_signal(s->mutex);
   }
}

void signal(sem *s) {
   bin_wait(s->mutex);
   s->value++;
   if(s->value <= 0)
      bin_signal(s->wait);
   bin_signal(s->mutex);
}
```

Deadlocks

```
// P1              // P2
wait(S);           wait(Q);
wait(Q);           wait(S);
…                  …
signal(S);         signal(Q);
signal(Q);         signal(S);
```

Deadlocks

```
// P1
wait(S);
wait(Q);
…
signal(S);
signal(Q);
```

```
// P2
wait(Q);
wait(S);
…
signal(Q);
signal(S);
```

```
P1: wait(S);
P2: wait(Q);
P1: wait(Q);
P2: wait(S);
// pwned
```

Mutex

- Semanticamente: Semáforo de tamanho 1
- [Geralmente] Apenas o processo/thread que adquire pode liberar
  - Erro em pthreads
  - ReentrantLock em Java lança exception
- lock/unlock

Chamadas Mutex em pthreads

```
//Cria mutex
int pthread_mutex_init(pthread_mutex_t *m, …)

//Trava mutex
int pthread_mutex_lock(pthread_mutex_t *m)

//Tenta travar e retorna imediatamente caso falhe
int pthread_mutex_trylock(pthread_mutex_t *m)

//Desbloqueia mutex
int pthread_mutex_unlock(pthread_mutex_t *m)
```

Como resolver o problema do contador?

```
int main() {
  pthread_mutex_t mutex;
  pthread_mutex_init(&mutex, NULL);
  // . . . cria e dispara as 2 threads passando o mutex para cada
}

void *mythread(void* arg) {
  pthread_mutex_t *mutex = (pthread_mutex_t*) arg;
  for (int i = 0; i < 1e7; i++) {
    pthread_mutex_lock(mutex);
    counter = counter + 1;
    pthread_mutex_unlock(mutex);
  }
}
```

Até agora

- Temos soluções para problemas de **mutual exclusion**
  - Spinlocks, semáforos, mutexes
- Temos soluções para **performance**
  - sleep/wakeup no semáforo
  - Menos desperdício de CPU
- Como resolver problemas de **ordem**?
  - Pi tem que executar antes do que Pj
  - Exemplo: A tem que ser impresso antes do que B

Sincronização com semáforos de tamanho 0

Processo Pi,
task inicializada com 0

```
// do something
signal(&task);
```

Processo Pj depende da
tarefa realizada por Pi

```
wait(&task);
// do something else
```

UFMG - ICEx
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

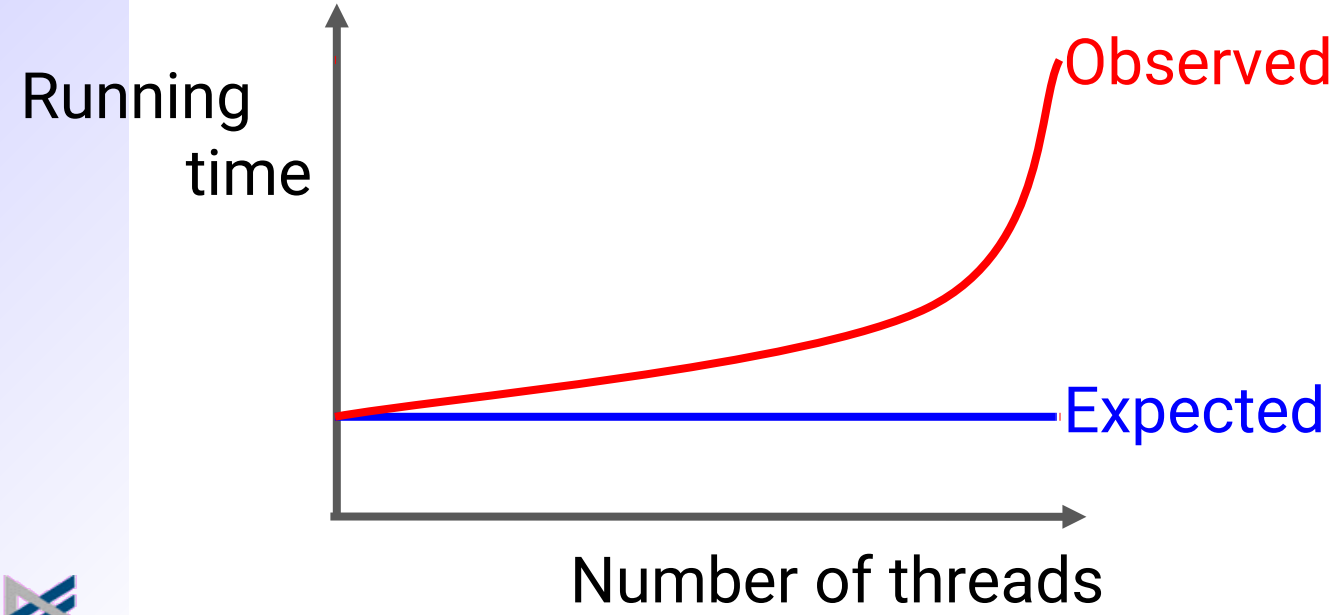
Condições

- Primitivas de wait e notify
  - wait libera a CPU e entra em uma fila de espera
  - notify avisa para alguém que está esperando
- Associados a um mutex

```
int pthread_cond_init(pthread_cond_t *cond, …);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *m);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Notas Finais

- Locks/mutex/semáforo etc etc são caros!



UFMG - ICEx
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Problemas Clássicos

Produtores Consumidores

- Fila Limitada de tamanho N
- Produtor trava quando a fila tem N elementos
- Consumidor trava quando a fila tem 0 elementos

Produtores Consumidores: Semáforos

```
sem_t *mutex = 1
sem_t *empty = BUFSZ;
sem_t *full = 0;
```

Producer:

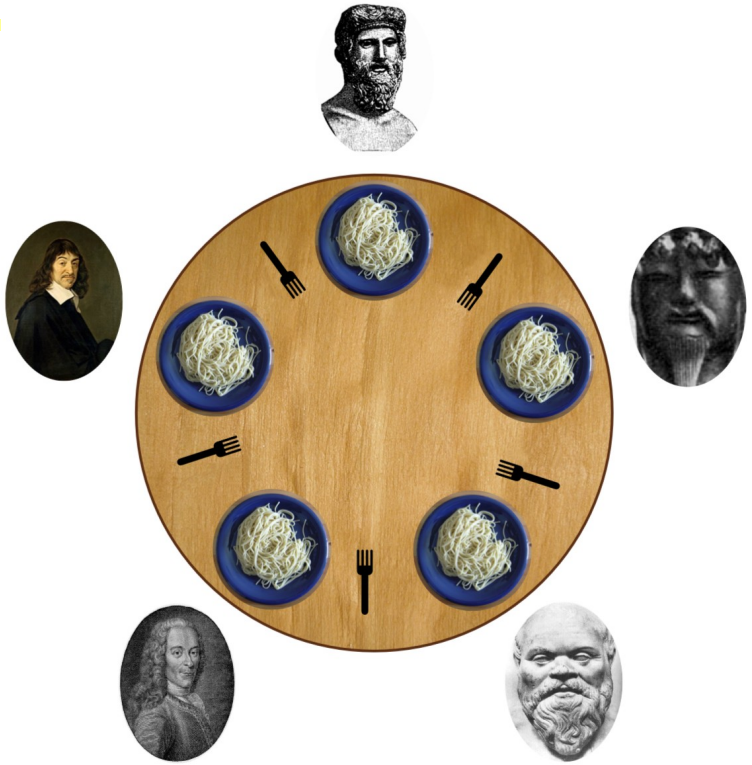
```
do {
   // generate item
   wait(empty);
   wait(mutex);
   // item → buffer
   signal(mutex);
   signal(full);
} while (1);
```

Consumer:

```
do {
   wait(full);
   wait(mutex);
   // item ← buffer
   signal(mutex);
   signal(empty);
   // process item
} while (1);
```

Jantar dos Filósofos

```
do {
    pickUpForks();
        eat();
    dropForks();
    think();
} while (1);
```



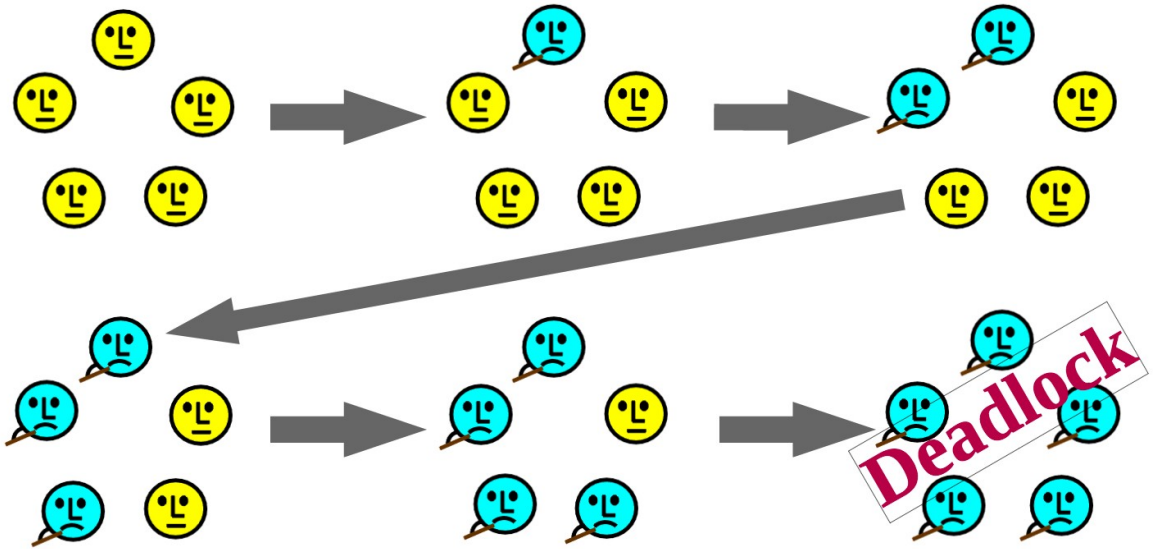
Uma Solução

Philosopher i:

```
sem_t **chopstick = [1, 1, 1, …]
do {
   // think
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
   // eat
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
} while (1);
```

Jantar dos Filósofos

- Para resolver precisamos tirar ciclos
  - Causam deadlocks



Jantar dos Filósofos

- Uma Solução
  - Pegar o garfo de menor ID
  - Vai ser o garfo da esquerda para todos menos o último filósofo
  - Sempre alguém vai conseguir comer dessa forma
  - [No fim] Removemos o ciclo de dependências dos garfos

Jantar dos Filósofos

- Uma Solução
  - Pegar o garfo de menor ID
  - Vai ser o garfo da esquerda para todos menos o último filósofo
  - Sempre alguém vai conseguir comer dessa forma
  - [No fim] Removemos o ciclo de dependências dos garfos

- Outra Solução
  - Garçon
  - Trava todos os garfos
  - Escolhe quem pode comer

Leitores Escritores

- Imagine um espaço de memória que pode ser lido e escrito
  - um DB por exemplo
- Apenas 1 escritor pode editar a memória
- N-Leitores podem ler
  - Leituras não causam problemas
- A leitura não pode ser corrompida por um escritor

Leitores Escritores

```
sem_t *mutex = 1
sem_t *writer = 1;
int rdcnt = 0;
```

Writer:

```
do {
  // other code
  wait(writer);
  // modify data
  signal(writer);
} while (1);
```

Reader:

```
do {
  wait(mutex);
  rdcnt++;
  if(rdcnt == 1) wait(writer);
  signal(mutex);
  // read data
  wait(mutex);
  rdcnt--;
  if(rdcnt == 0) signal(writer);
  signal(mutex);
  // other code
} while (1);
```

Referências

- Arpaci-Dusseau, Arpaci-Dusseau; OSTEP, v0.91
  - Chapters 25, 26, 27, 28, 30, 31
- Silberschatz, Galvin, Gagne; Operating System Fundamentals, 9th Edition
  - Chapter 5
- Tanenbaum; Modern Operating Systems; 4th Edition
  - Chapter 2